



X-KAAPI: a Multi Paradigm Runtime for Multicore Architectures

Thierry Gautier*, Fabien Lementec*, Vincent Faucher†, Bruno Raffin*

*INRIA, Grenoble, France

†CEA, DEN, DANS, DM2S, SEMT, DYN, Gif-sur Yvette, France

Thierry Gautier
thierry.gautier@inrialpes.fr
MOAIS, INRIA, Grenoble

Parallel architecture

- Complex architecture

- ▶ Computing resources

- CPU, GPU, ...

- ▶ Memory

- hierarchical memory (register, L1, L2, L3, main memory)
 - private / shared cache

- ▶ Interconnection network

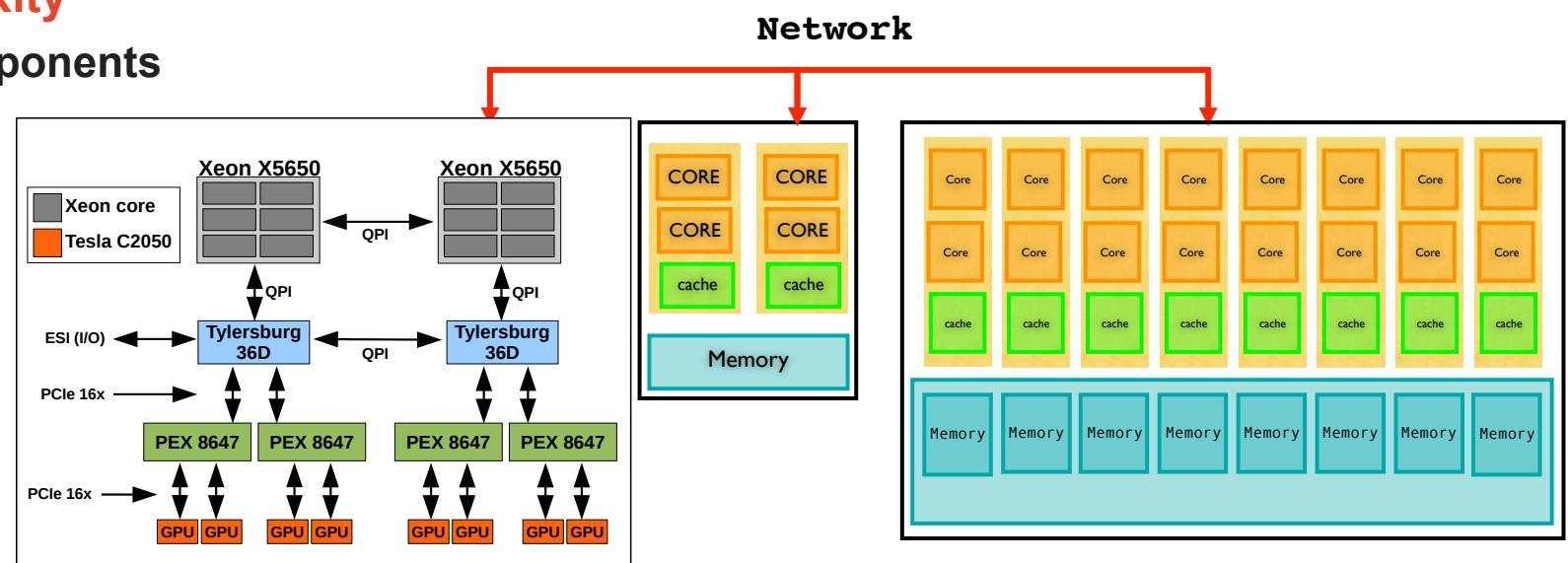
- between several cores & memory

→ High complexity

- ▶ million of components

- ▶ heterogeneity

- memory
 - processor

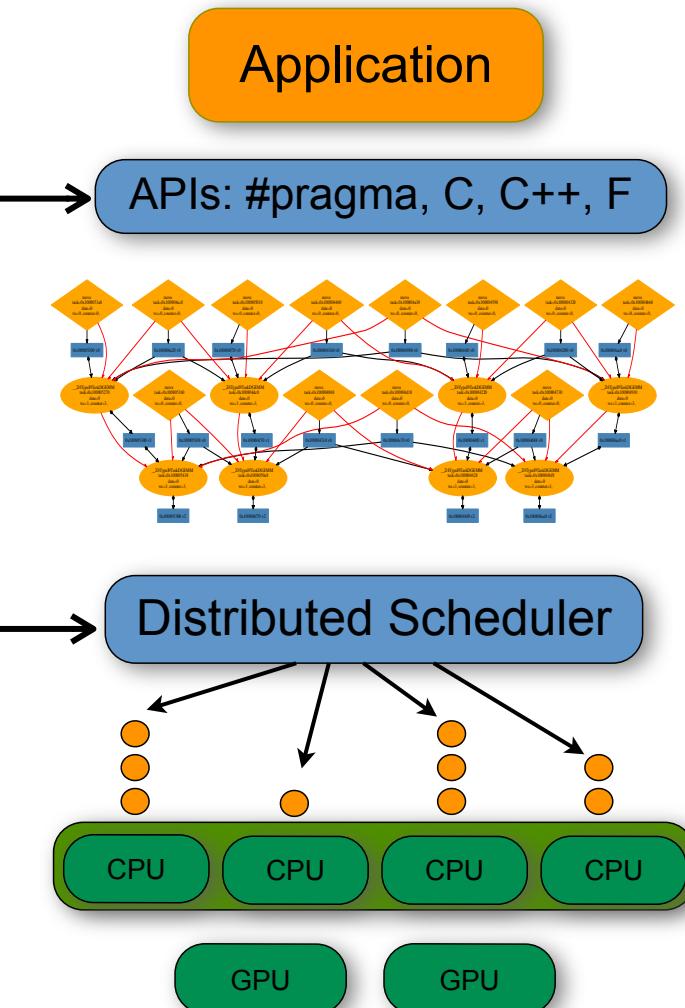


Goal: Write Once, Run Anywhere

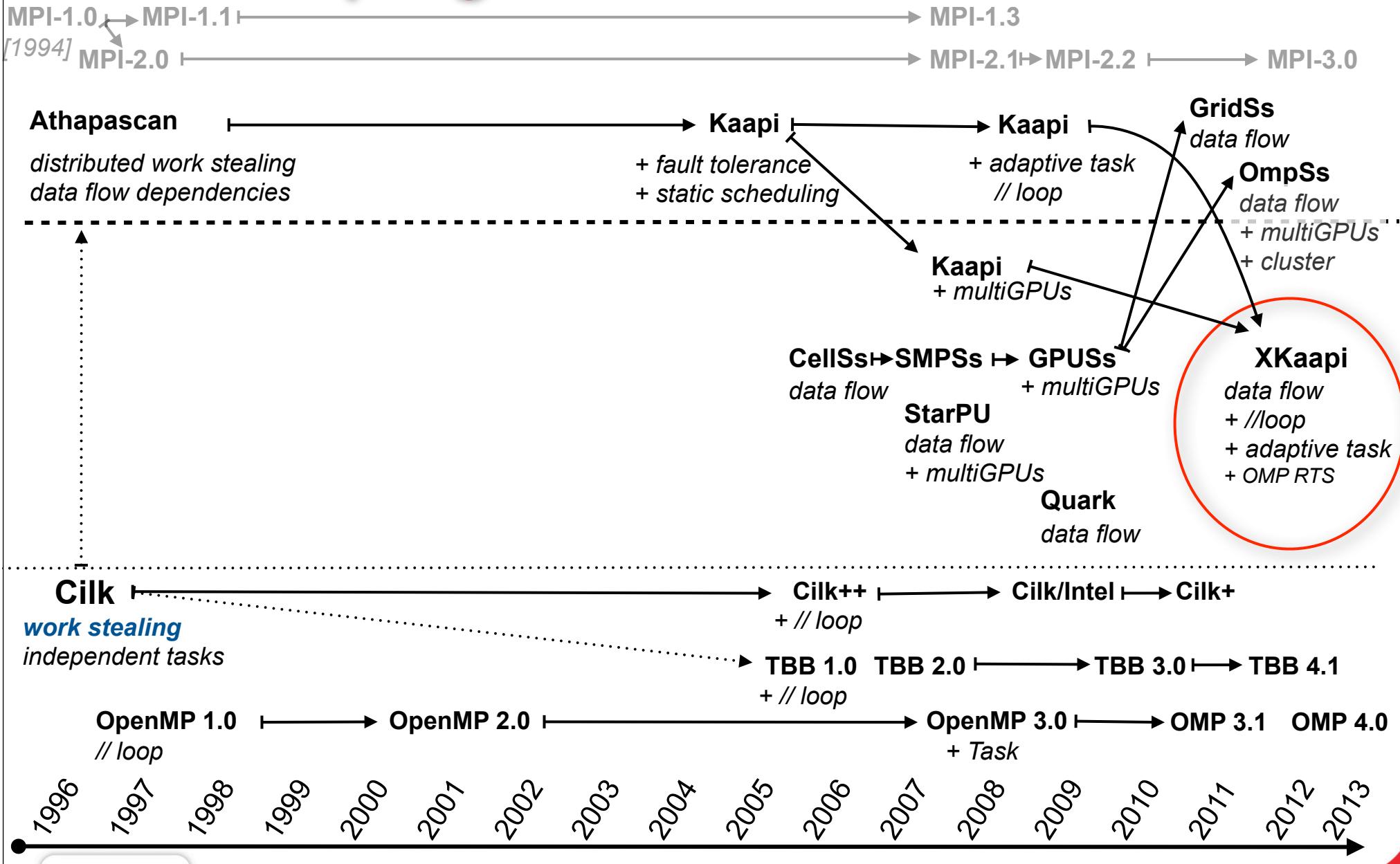
- Provide performance guarantee of application
 - On multiple parallel architectures
 - With dynamic variation (OS jitter, application load)

- Two steps solution

- Definition of a programming model → **APIs: #pragma, C, C++, F**
 - Task based
 - recursive task, adaptive task
 - Data flow dependencies
 - computed at runtime
- Efficient scheduling algorithms → **Distributed Scheduler**
 - Work stealing based with heuristic
 - HEFT, DualApproximation, ...
 - Theoretical analysis of performance



How to program such architecture ?



Outline

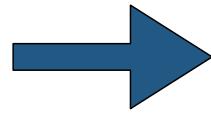
- Introduction
- Overview of Kaapi parallel programming model
 - Scheduling tasks with data flow dependencing
 - XKaapi's on-demand task creation
- Evaluations
 - Micro benchmarks
 - EPX parallelization
- Conclusions

Data flow dependencies

- Using code annotation

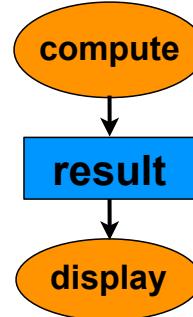
```
void main()
{
    /* data result is produced */
    compute( input, &result );

    /* data result is consumed */
    display( &result );
}
```



```
void main()
{
#pragma kaapi task read(input) write(result)
    compute( input, &result );

#pragma kaapi task read(result)
    display( &result );
}
```



- Other APIs: C, C++, Fortran

- Task ~ OpenMP structured block

- assumption: no side effect, description of access mode

- Related work

- StarPU [Bordeaux, France], OmpSS [BSC, Spain], Quark [UTK]
 - and new standard OpenMP-4.0 !

XKaapi programming example

```
#include <cblas.h>
#include <clapack.h>
void Cholesky( double* A, int N, size_t NB )
{
    for (size_t k=0; k < N; k += NB)
    {
        clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );
        for (size_t m=k+ NB; m < N; m += NB)
        {
            cblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit,
                           NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );
        }
        for (size_t m=k+ NB; m < N; m += NB)
        {
            cblas_dsyrk ( CblasRowMajor, CblasLower, CblasNoTrans,
                           NB, NB, -1.0, &A[m*N+k], N, 1.0, &A[m*N+m], N );
            for (size_t n=k+NB; n < m; n += NB)
            {
                cblas_dgemm ( CblasRowMajor, CblasNoTrans, CblasTrans,
                               NB, NB, NB, -1.0, &A[m*N+k], N, &A[n*N+k], N, 1.0, &A[m*N+n], N );
            }
        }
    }
}
```

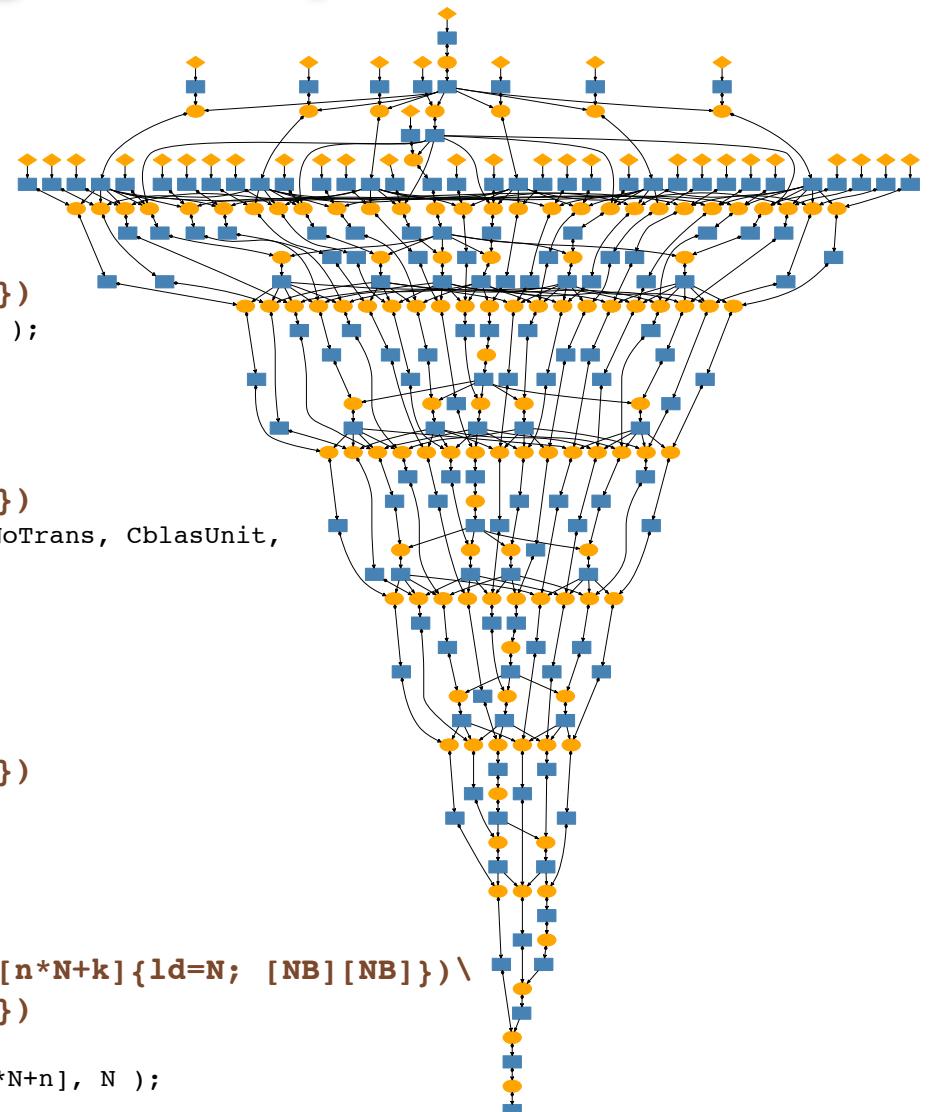
XKaapi programming example

```
#include <cblas.h>
#include <clapack.h>
void Cholesky( double* A, int N, size_t NB )
{
    for (size_t k=0; k < N; k += NB)
    {
        #pragma kaapi task readwrite(&A[k*N+k]{ld=N; [NB][NB]}) \
            clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );

        for (size_t m=k+ NB; m < N; m += NB)
        {
            #pragma kaapi task read(&A[k*N+k]{ld=N; [NB][NB]}) \
                readwrite(&A[m*N+k]{ld=N; [NB][NB]}) \
                    cblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit,
                        NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );
        }

        for (size_t m=k+ NB; m < N; m += NB)
        {
            #pragma kaapi task read(&A[m*N+k]{ld=N; [NB][NB]}) \
                readwrite(&A[m*N+m]{ld=N; [NB][NB]}) \
                    cblas_dsyrk ( CblasRowMajor, CblasLower, CblasNoTrans,
                        NB, NB, -1.0, &A[m*N+k], N, 1.0, &A[m*N+m], N );

            for (size_t n=k+NB; n < m; n += NB)
            {
                #pragma kaapi task read(&A[m*N+k]{ld=N; [NB][NB]}, &A[n*N+k]{ld=N; [NB][NB]}) \
                    readwrite(&A[m*N+n]{ld=N; [NB][NB]}) \
                    cblas_dgemm ( CblasRowMajor, CblasNoTrans, CblasTrans,
                        NB, NB, NB, -1.0, &A[m*N+k], N, &A[n*N+k], N, 1.0, &A[m*N+n], N );
            }
        }
    }
}
```



Main Characteristics of XKaapi

- Parallelism is explicit, task based, with data flow dependencies

- Task's creation is a non blocking operation
- Dependencies between tasks = Data flow dependencies
 - Computed at runtime during workstealing requests
≠ StarPU, OmpSS, Quark = computed during task's creation

- Scheduling

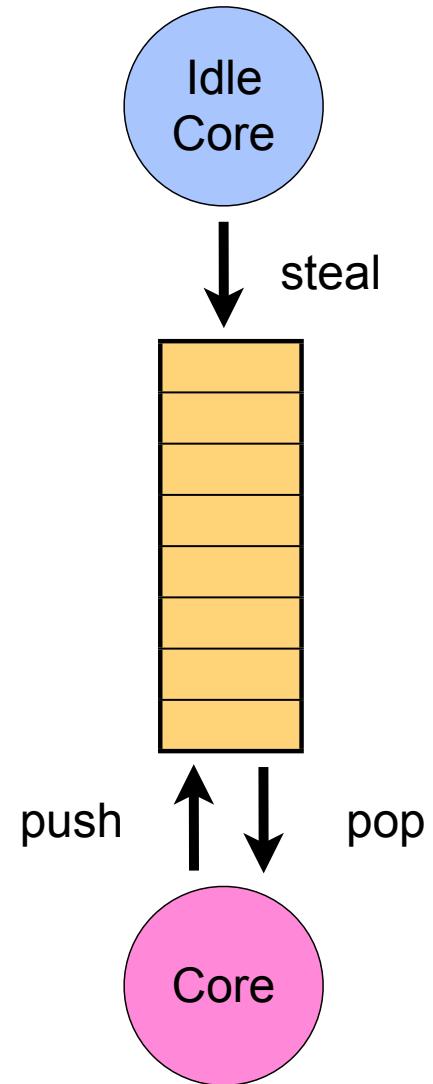
- by work stealing
 - Cilk's like performance guarantee
 - $T_p = O(T_1/p + T_\infty)$
 - Number of steal requests $O(p T_\infty)$
 - + heuristics for data locality
- + others: ETF, HETF, DualApproximation

- Target architectures

- heterogeneous architecture: multi-CPUs / multi-GPUs
- many-core: Intel Xeon Phi

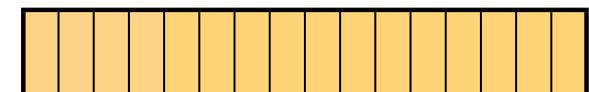
The way XKaapi executes tasks

- One “worker thread” per core
 - Able to execute XKaapi fine-grain tasks
 - Holds a queue of tasks
 - Related to sequential C stack of activation frames
 - T.H.E. low overhead protocol, lock in rare case
- Task creation is cheap !
 - Reduces to pushing C function pointer + its arguments into the worker thread queue
 - ~ 10 cycles / tasks on AMD Many Cours processors
 - Recursive tasks are welcome
- Work-stealing based scheduling
 - Cilks's work first principle
 - Work-stealing algorithm = plug-in
 - Default: steal a task from a randomly chosen queue



On-demand task creation with XKaapi

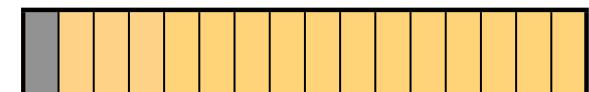
- Adaptive tasks in XKaapi
 - Adaptative tasks can be split at run time to create new tasks
 - Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution
- Example of use: the XKaapi *for_each* construct
 - A general-purpose parallel loop
 - A task == a range of iterations to compute
 - Execution model
 - Initially, one task in charge of the whole range



$T_1 : [0 - 15]$

On-demand task creation with XKaapi

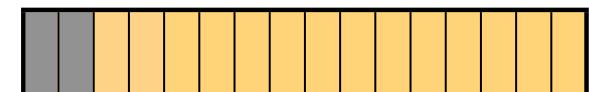
- Adaptive tasks in XKaapi
 - Adaptative tasks can be split at run time to create new tasks
 - Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution
- Exemple of use: the XKaapi *for_each* construct
 - A general-purpose parallel loop
 - A task == a range of iterations to compute
 - Execution model
 - Initially, one task in charge of the whole range



$T_1 : [0 - 15]$

On-demand task creation with XKaapi

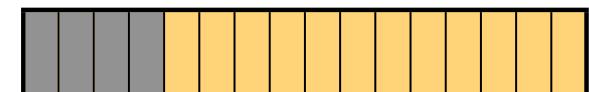
- Adaptive tasks in XKaapi
 - Adaptative tasks can be split at run time to create new tasks
 - Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution
- Exemple of use: the XKaapi *for_each* construct
 - A general-purpose parallel loop
 - A task == a range of iterations to compute
 - Execution model
 - Initially, one task in charge of the whole range



$T_1 : [0 - 15]$

On-demand task creation with XKaapi

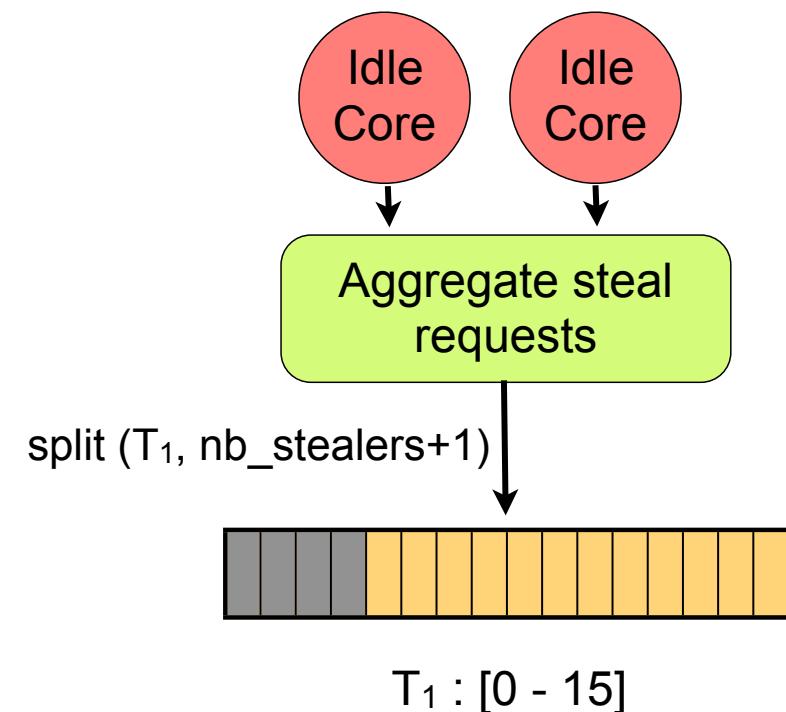
- Adaptive tasks in XKaapi
 - Adaptative tasks can be split at run time to create new tasks
 - Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution
- Exemple of use: the XKaapi *for_each* construct
 - A general-purpose parallel loop
 - A task == a range of iterations to compute
 - Execution model
 - Initially, one task in charge of the whole range



$T_1 : [0 - 15]$

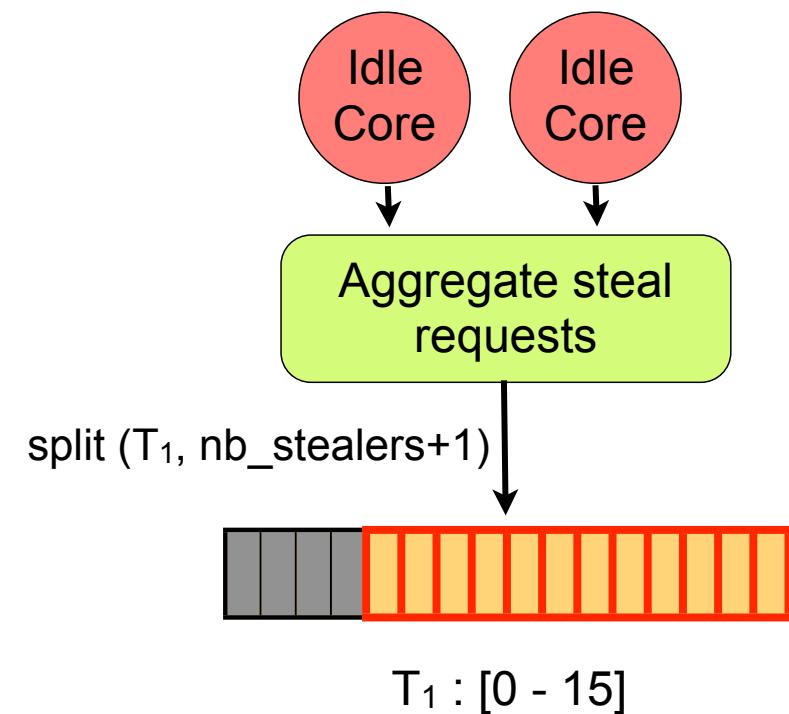
On-demand task creation with XKaapi

- Adaptive tasks in XKaapi
 - Adaptative tasks can be split at run time to create new tasks
 - Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution
- Exemple of use: the XKaapi *for_each* construct
 - A general-purpose parallel loop
 - A task == a range of iterations to compute
 - Execution model
 - Initially, one task in charge of the whole range
 - Idle cores post steal requests and trigger the «split» operation to generate new tasks



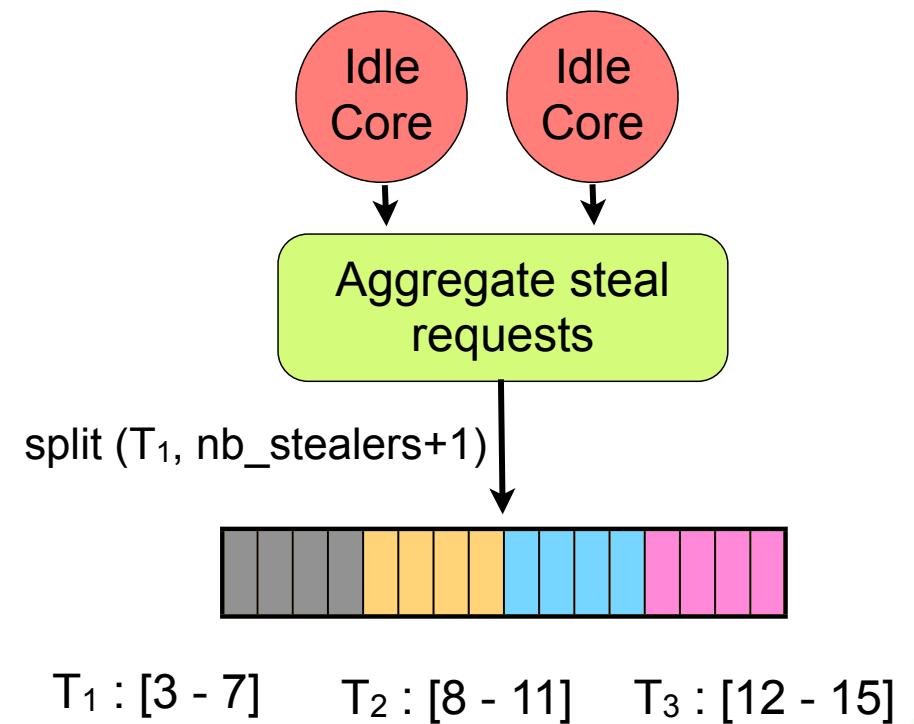
On-demand task creation with XKaapi

- Adaptive tasks in XKaapi
 - Adaptative tasks can be split at run time to create new tasks
 - Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution
- Exemple of use: the XKaapi *for_each* construct
 - A general-purpose parallel loop
 - A task == a range of iterations to compute
 - Execution model
 - Initially, one task in charge of the whole range
 - Idle cores post steal requests and trigger the «split» operation to generate new tasks



On-demand task creation with XKaapi

- Adaptive tasks in XKaapi
 - Adaptative tasks can be split at run time to create new tasks
 - Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution
- Exemple of use: the XKaapi *for_each* construct
 - A general-purpose parallel loop
 - A task == a range of iterations to compute
 - Execution model
 - Initially, one task in charge of the whole range
 - Idle cores post steal requests and trigger the «split» operation to generate new tasks



Outline

- Introduction
- Overview of Kaapi parallel programming model
 - Scheduling tasks with data flow dependencing
 - XKaapi's on-demand task creation
- Evaluations
 - Micro benchmarks
 - EPX parallelization
- Conclusions

Overhead of task management [AMD48]

Cilk+

```
long fib(long n)
{
    if (n < 2)
        return (n);
    else {
        long x, y;
        x = cilk_spawn
        y = fib(n - 2);
        cilk_sync;
        return (x + y);
    }
}
```

Kaapi

OpenMP

```
void fibonacci(
{
    if (n<2)
        *result =
    else
    {
        long r1,r2;
#pragma omp ta
    }
}
```

TBB

```
struct FibContinuation: public tbb::task {
    long* const sum;
    long x, y;
    FibContinuation( long* sum_ ) : sum(sum_) {}
    tbb::task* execute() {
        *sum = x+y;
        return NULL;
    }

    struct FibTask: public tbb::task {
        long n;
        long * sum;
        FibTask( const long n_, long * const sum_ ) :
            n(n_), sum(sum_)
    }
}
```

```
void fibonacci(long* result, const long n)
{
    if (n<2)
        *result = n;
    else
    {
        long r1,r2;
#pragma kaapi task write(&r1)
        fibonacci( &r1, n-1 );
        fibonacci( &r2, n-2 );
#pragma kaapi sync
        *result = r1 + r2;
    }
}
```

ntinuation()) FibContinuation(sum);
allocate_child()) FibTask(n-1,&c.y);
);

two children".



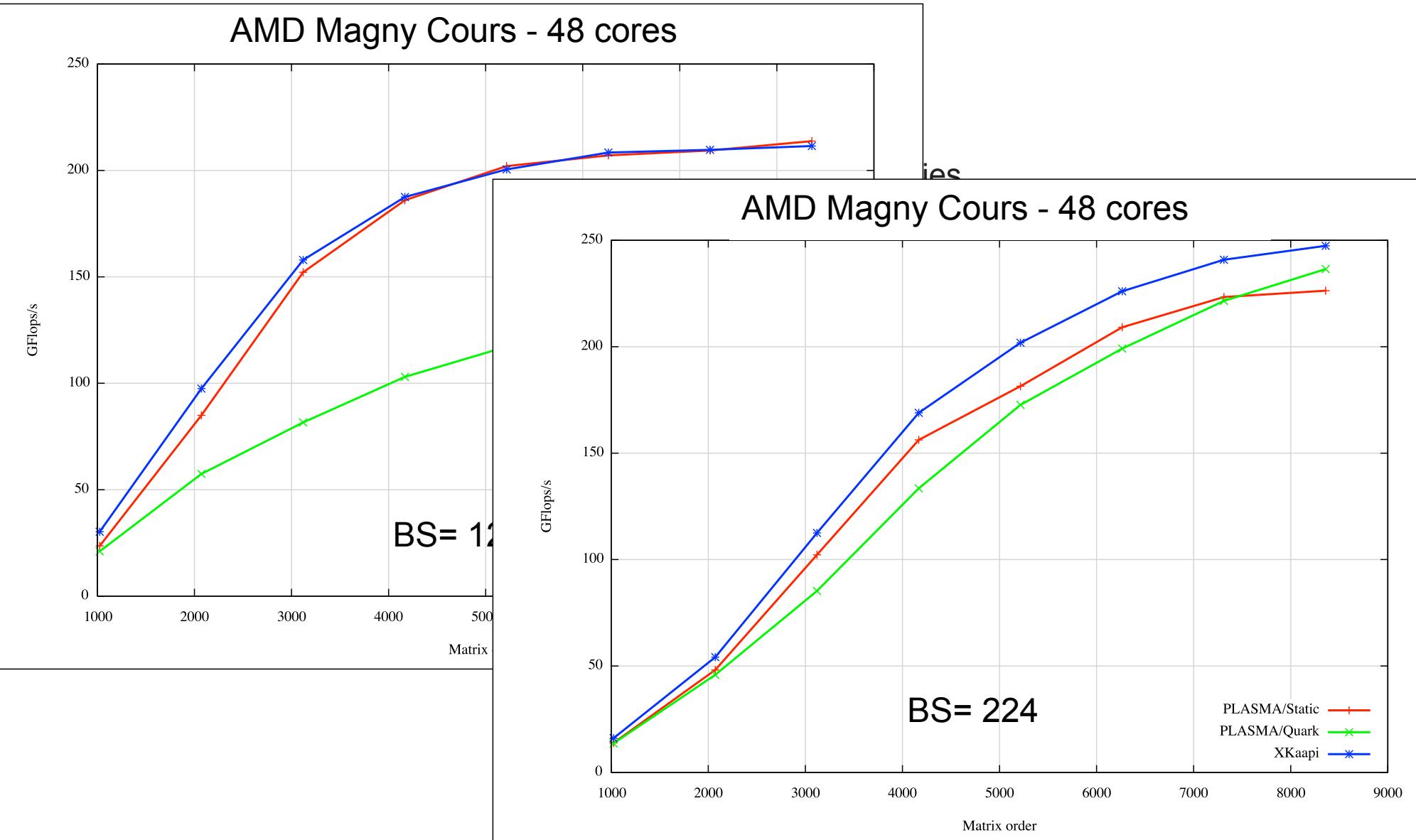
Overhead of task management [AMD48]

- Fibonacci (35) naive recursive computation
- AMD Many Cours, 2.2GHz, 48 cores, 256GB main memory

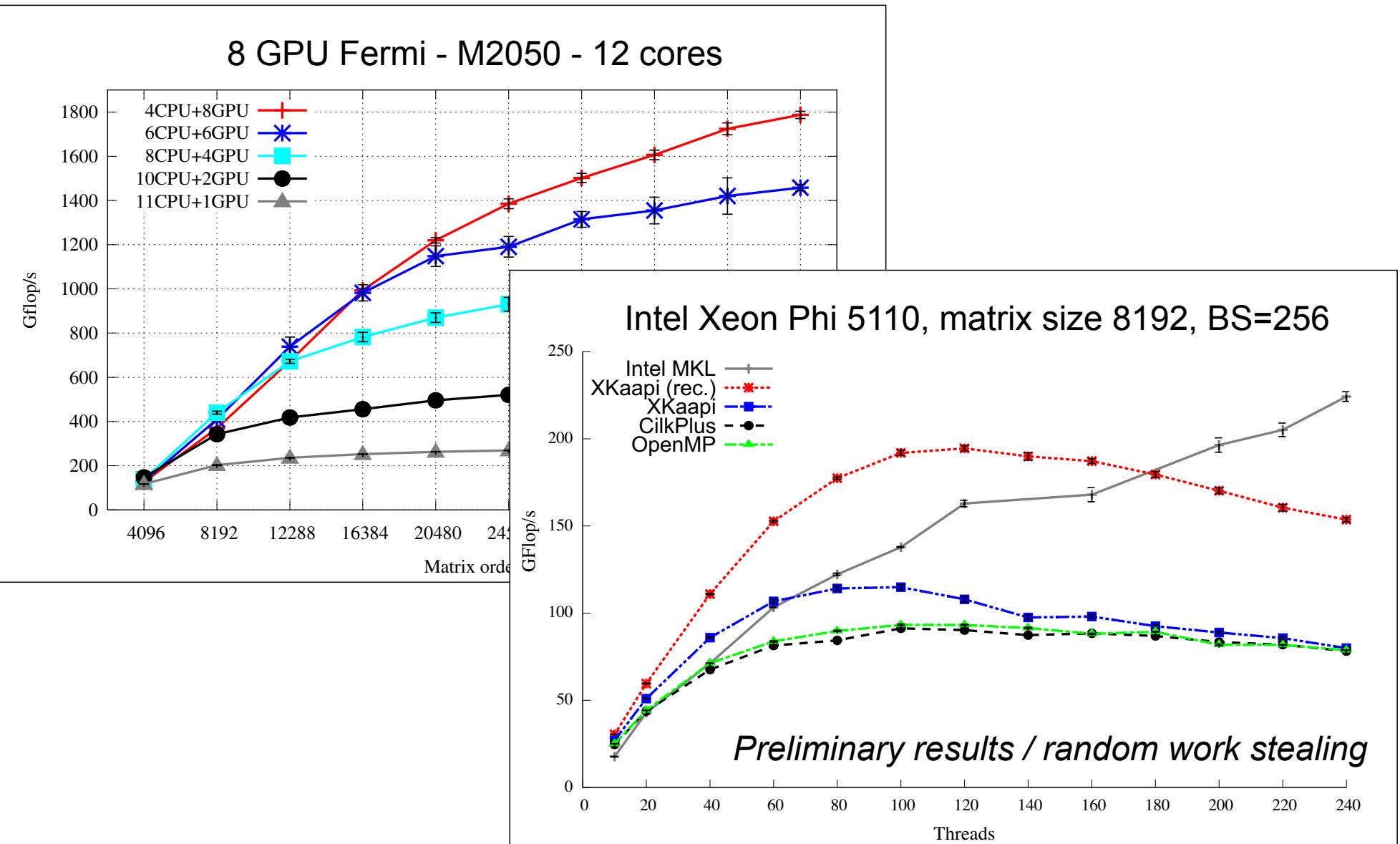
Serial	Cilk+	TBB (4.0)	OpenMP (gcc)	XKaapi
0.0905s (x 1)	1.063 (x 11.7)	2.356s (x 26)	2.429s (x 27)	0.728s (x 8)

#Cores	Cilk+ (s)	TBB 4.0 (s)	XKaapi (s)	OpenMP (gcc) (s)
1	1.063	2.356	0.728	2.43
8	0.127	0.293	0.094	51.06
16	0.065	0.146	0.047	104.14
32	0.035	0.072	0.024	NO TIME
48	0.028	0.049	0.017	NO TIME

Dense Cholesky factorization

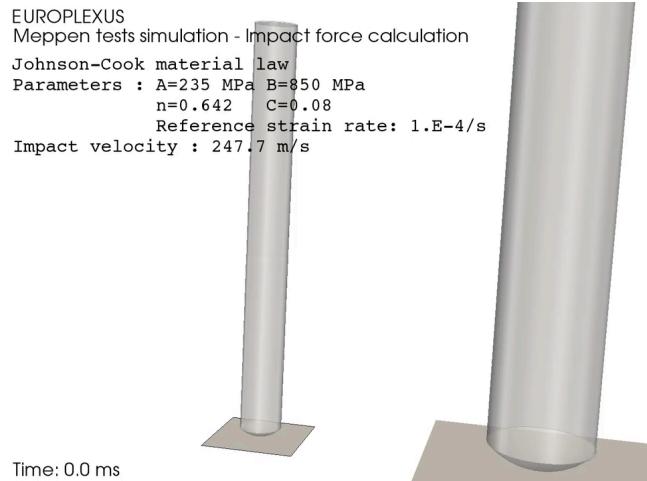


Dense Cholesky factorization

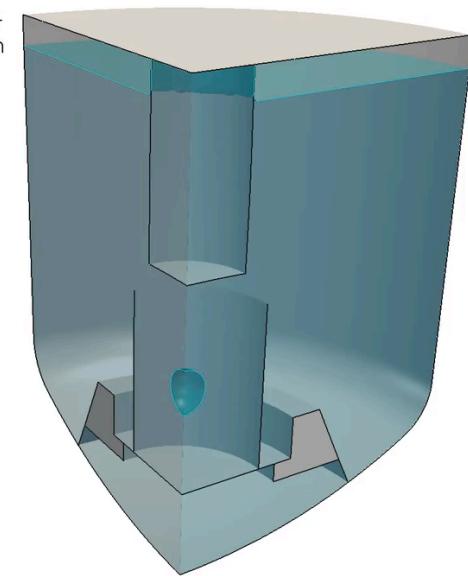


Parallelization of EPX

- Multicore parallelization of EPX (EUROPLEXUS) code [CEA - IRC - EDF - ONERA]
 - ▶ ANR RepDyn funding.
- Fluid-Structure systems subjected to fast transient dynamic loading



EUROPLEXUS
Simulation of MARA10 experiment
ADCR material - VOFIRE algorithm

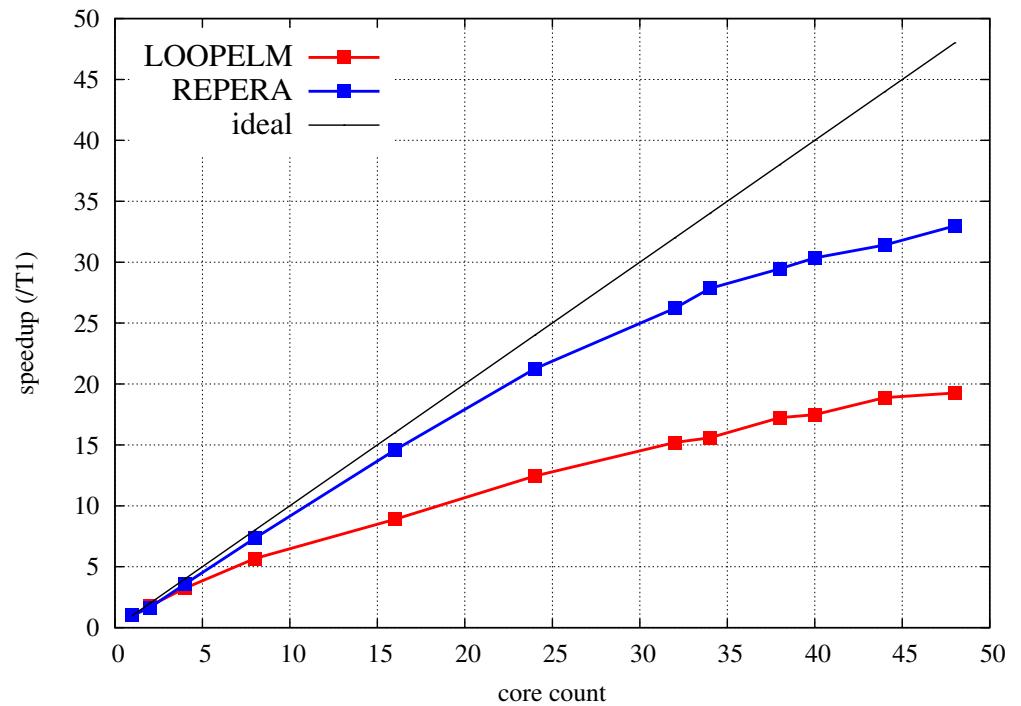
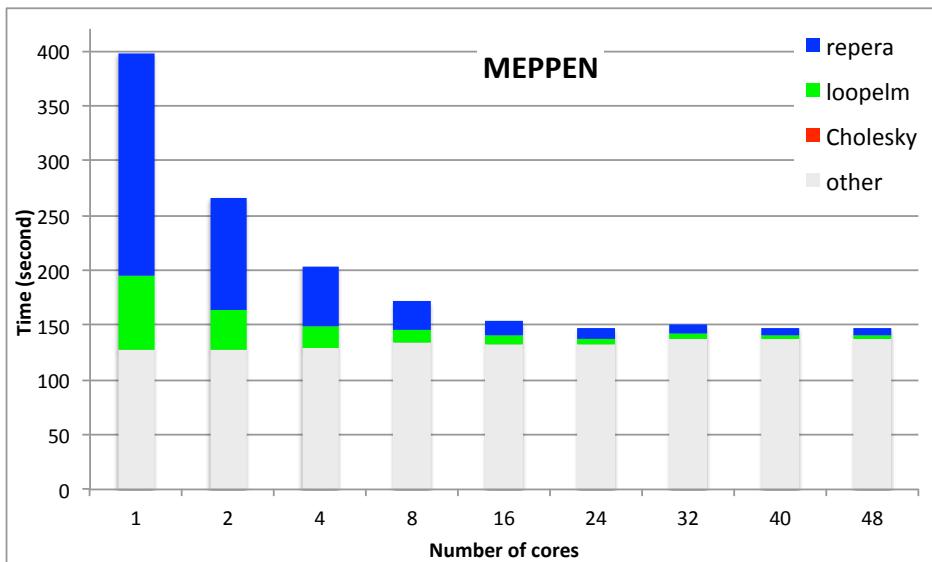


Parallelization of EPX

- Complex code
 - ▶ 600 000 lines of code (Fortran)
- Two main sources of parallelization (~70% of the computation)
 - ▶ Sparse Cholesky factorization
 - skyline representation
 - XKaapi Parallel program = dependent tasks with data flow dependencies
 - ▶ 2 Independent loops
 - LOOPELM:
 - iteration over finite elements to compute nodal internal forces
 - REPERA:
 - iteration for kinematic link detection
 - Parallelization = on-demand task creation through the XKaapi's parallel foreach functor
- Two instances
 - ▶ MEPPEN
 - ▶ MAXPLANE
- AMD Many Cours, 2.2Ghz, 48 cores, 256GB main memory

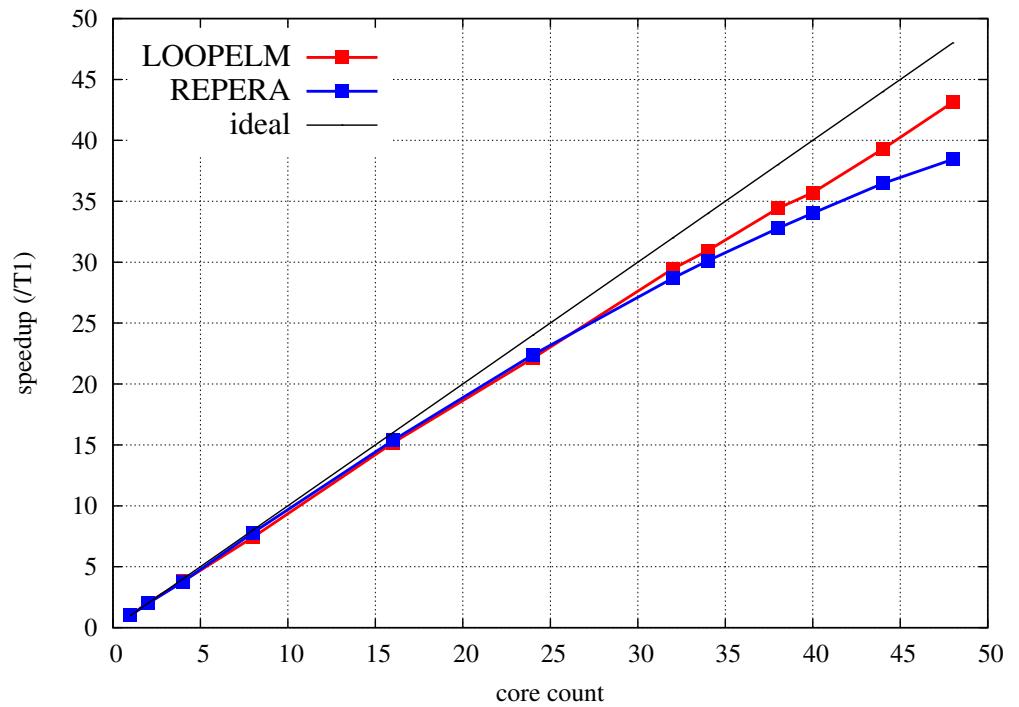
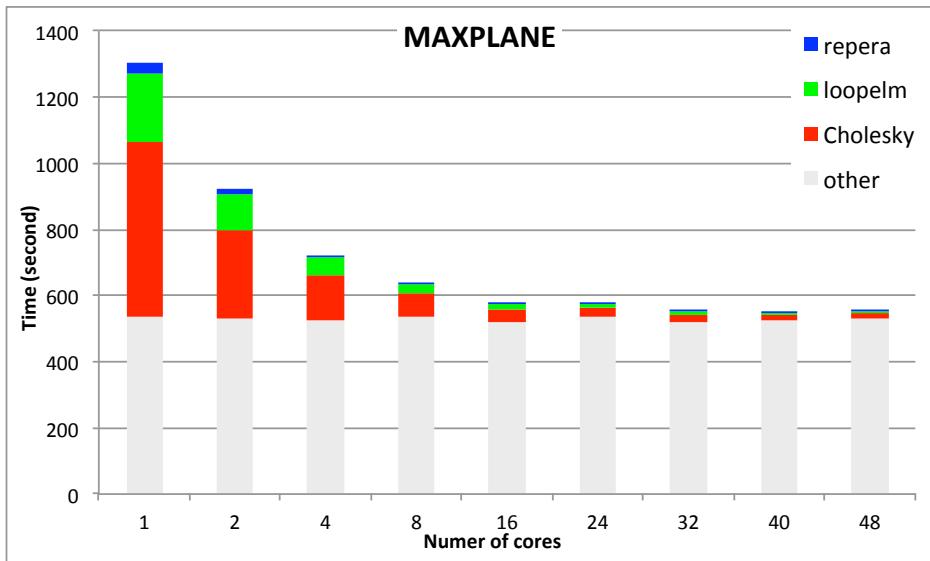
Case of study: MEPPEN

- Main characteristics
 - Most of the time in independent loops LOOPELM and REPERA
- AMD Many Cours, 2.2GHz, 48 cores, 256GB main memory



Case of study: MAXPLANE

- Main characteristics
 - Most of the time in sparse Cholesky factorization
- AMD Many Cours, 2.2GHz, 48 cores, 256GB main memory



Conclusions

- **XKaapi**

- Low overhead runtime for dependent tasks (data flow), scheduling algorithms
- Good performances on different architectures
 - multi-CPUs, multi-CPUs-multi-GPUs, Intel Xeon Phi

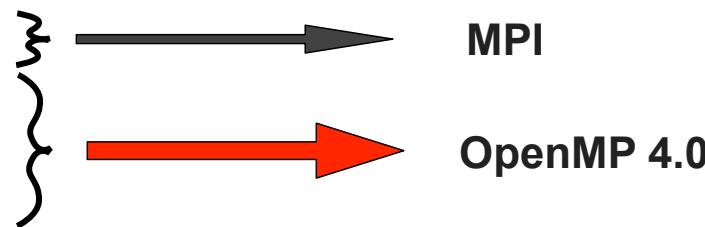
- **Improving parallelization of EPX**

- Analysis of the remainder sequential part
- Scalability on thousands of cores

- **Integration of our runtime under OpenMP-4.0 directives**

- Supercomputer = high performance network + multicores + accelerators
- Software stack

- Network:
- Multicores: OpenMP
- Accelerators: Cuda, OpenCL, OpenACC
- SIMD Units: Compiler or extension



- **More informations:** <http://kaapi.gforge.inria.fr>

- **[IWOMP2012]:**
 - replacement of libGOMP for OpenMP-3.1/GCC with better management of tasks
- **[IWOMP2013]:**
 - extension of loop scheduler based on “on-demand task creation”
- **[IPDPS2013]:**
 - multi-CPUs / multi-GPUs (12 cores machine + 8 GPUs)
- **[SBAC-PAD13]:**
 - Comparison Intel Xeon Phi / Intel Sandybridge with OpenMP, Cilk+ and Kaapi

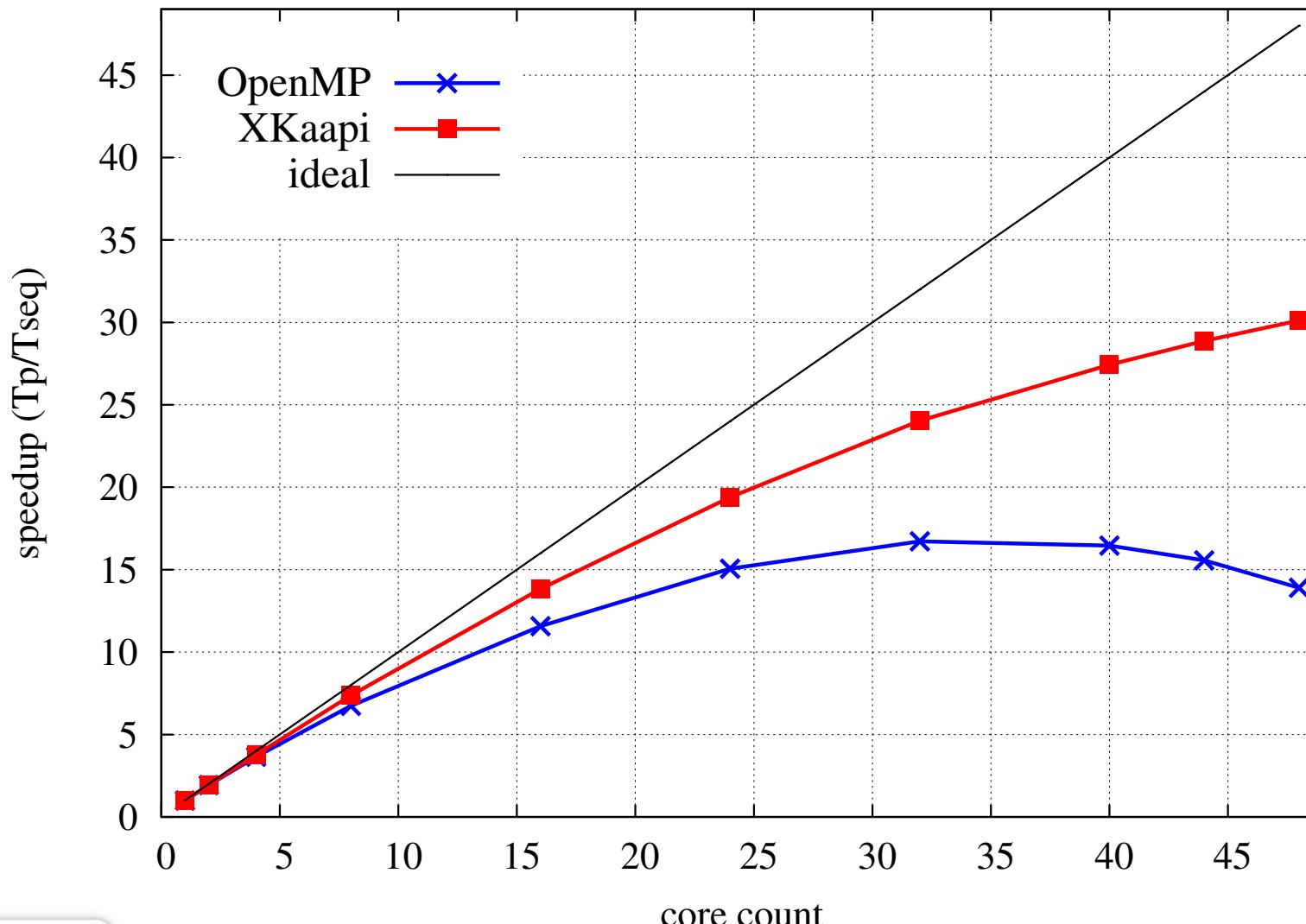
Thank you for your
attention !

<http://kaapi.gforge.inria.fr>

Sparse Cholesky Factorization (EPX)

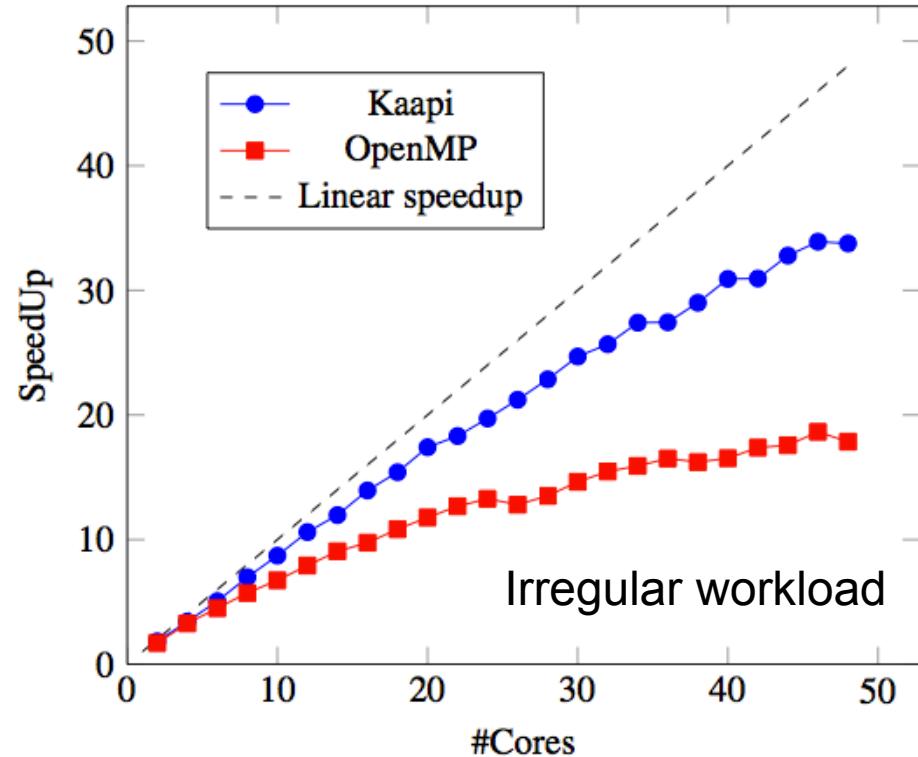
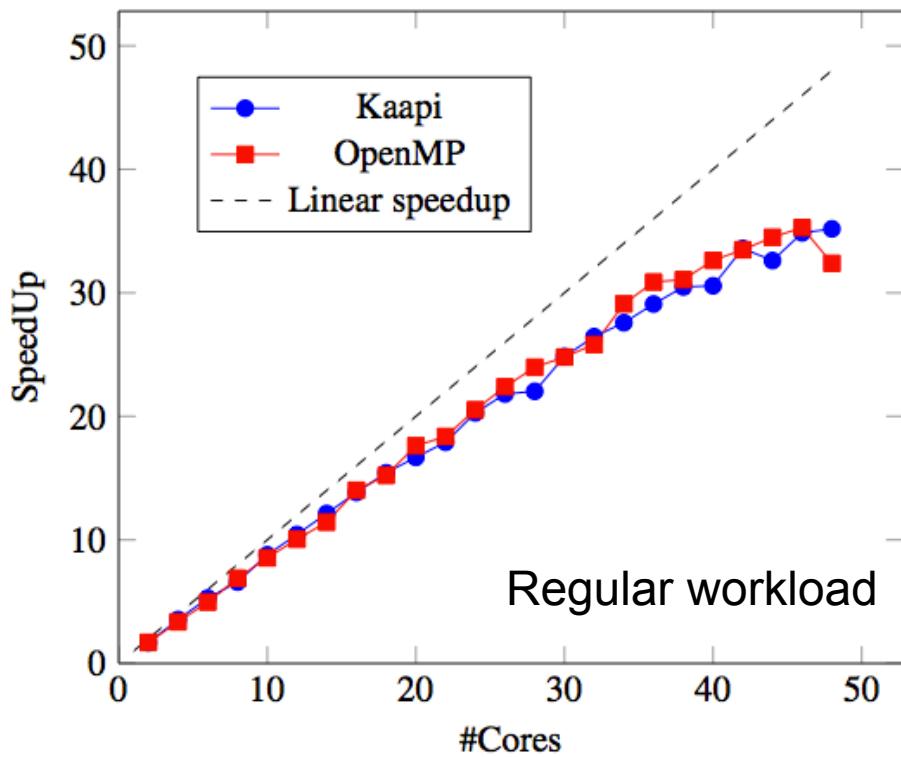
- OpenMP / XKaapi

- ▶ 59462 with 3.59% of non zero elements



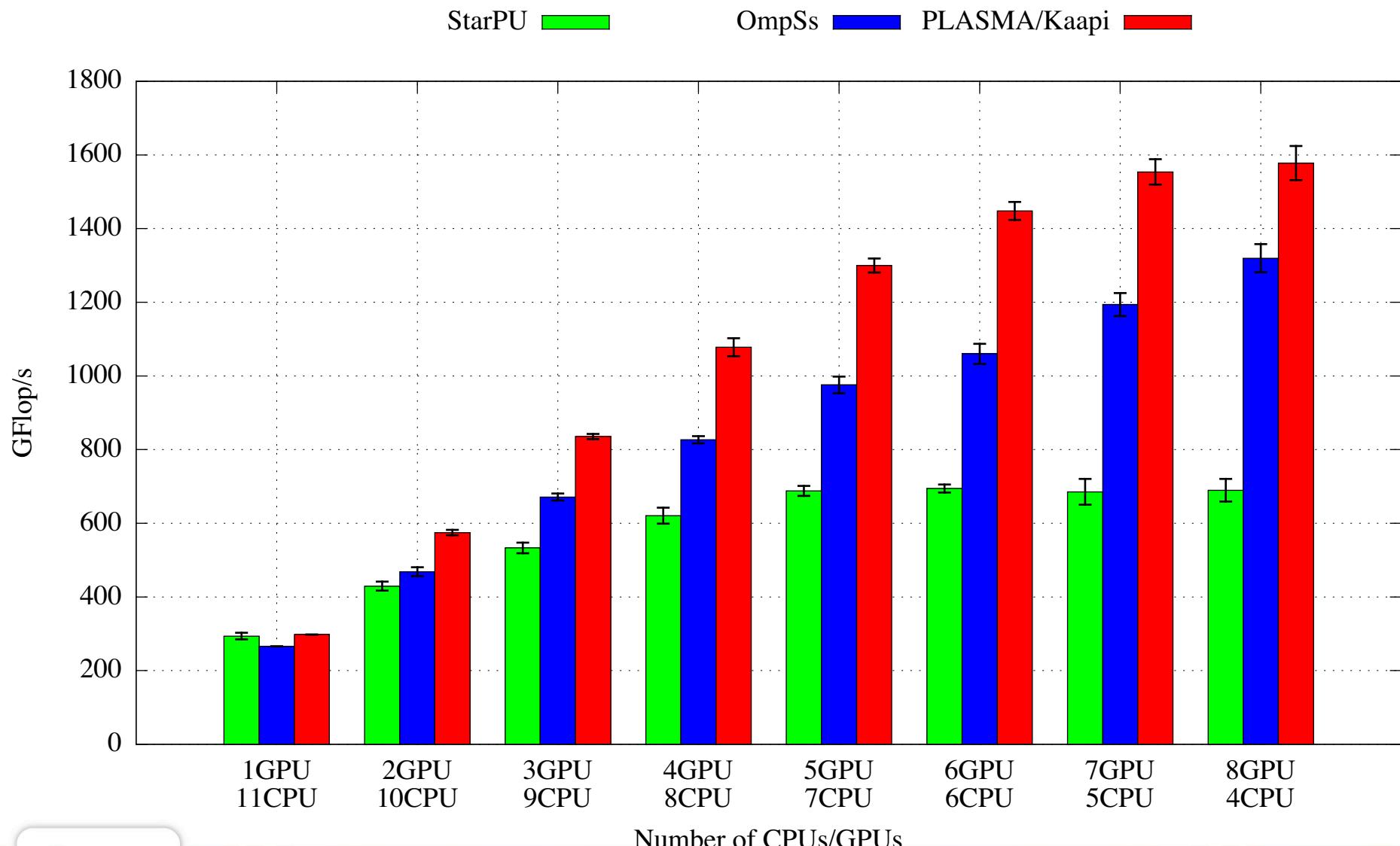
Performance evaluation: VTK filters

- Parallel version of the VTK visualization toolkit [M. Ettinger, MOAIS]
 - ▶ A framework to develop parallel applications for scientific visualization
 - ▶ A VTK «filters» == a computation performed on a 2D/3D scene
 - ▶ parallel loop, static OpenMP schedule



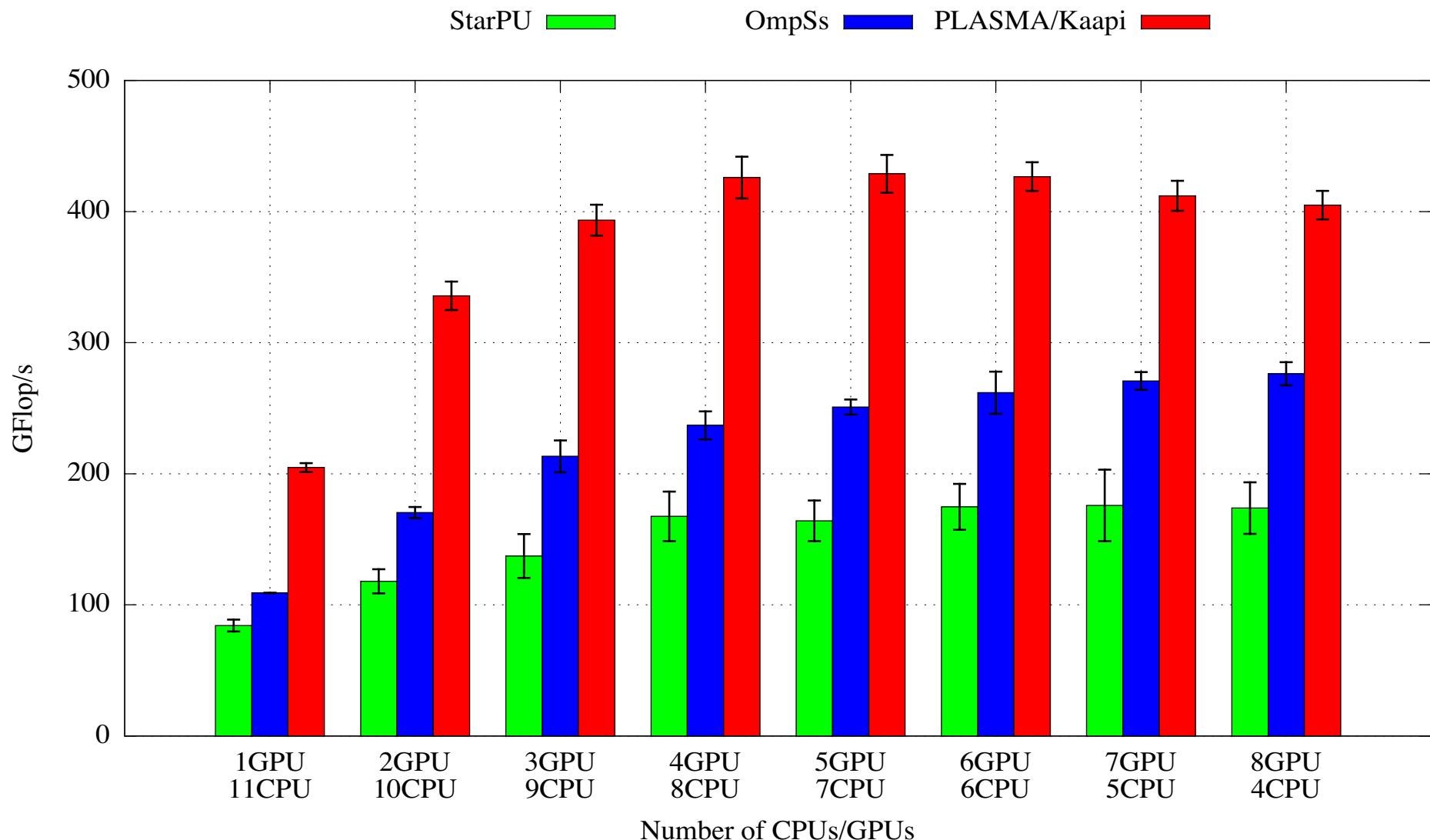
Comparison with OMPSS, StarPU

- DGEMM matrix size 10240, block size 1024



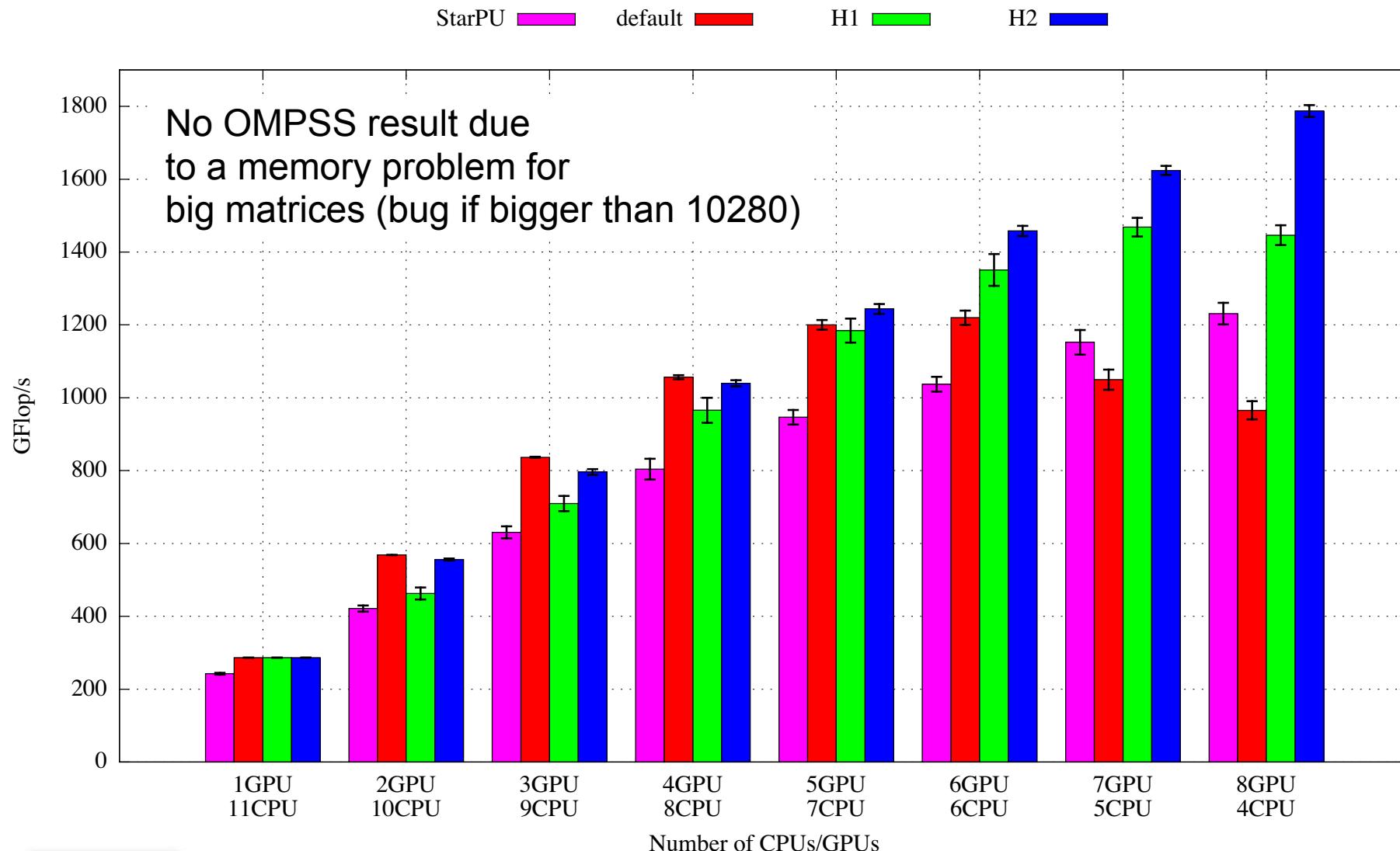
Comparison with OMPSS, StarPU

- DPOTRF matrix size 10240, block size 1024



Comparison with OMPSS, StarPU

- DPOTRF matrix size 40960, BS 1024



Improving OpenMP task implementation

- [IWOMP 2012]
- **Barcelona OpenMP Task Suite**
 - ▶ Using libKOMP = our libGOMP implementation (on top of XKaapi)
 - ▶ A set of representative benchmarks to evaluate OpenMP tasks implementations

Name	Arguments used	Domain	Summary
Alignment	prot100.aa	Dynamic programming	Aligns sequences of proteins
FFT	n=33,554,432	Spectral method	Computes a Fast Fourier Transformation
Floorplan	input.20	Optimization	Computes the optimal placement of cells in a floorplan
NQueens	n=14	Search	Finds solutions of the N Queens problem
MultiSort	n=33,554,432	Integer sorting	Uses a mixture of sorting algorithms to sort a vector
SparseLU	n=128 m=64	Sparse linear algebra	Computes the LU factorization of a sparse matrix
Strassen	n=8192	Dense linear algebra	Computes a matrix multiply with Strassen's method
UTS	medium.input	Search	Computes the number of nodes in an Unbalanced Tree

- **Evaluation platforms**
 - ▶ AMD48: 4x12 AMD Opteron (6174) cores
 - ▶ Intel32: 4x8 Intel Xeon (X7560) cores

- **Softwares**
 - ▶ gcc 4.6.2 + libGOMP
 - ▶ gcc 4.6.2 + libKOMP
 - ▶ icc 12.1.2 + Intel OpenMP runtime (KMP)

Running OpenMP BOTS with libKOMP

Speed-Up of BOTS kernels on the AMD48 platform

<i>kernel</i>	<i>libGOMP</i>	<i>libKOMP</i>	<i>Intel</i>
Alignment	38.8	40.0	37.0
FFT	0.5	12.2	12.0
Floorplan	27.6	32.7	29.2
NQueens	43.7	47.8	39.0
MultiSort	0.6	13.2	11.3
SparseLU	44.1	44.4	35.0
Strassen	20.8	22.4	20.5
UTS	0.9	25.3	15.0

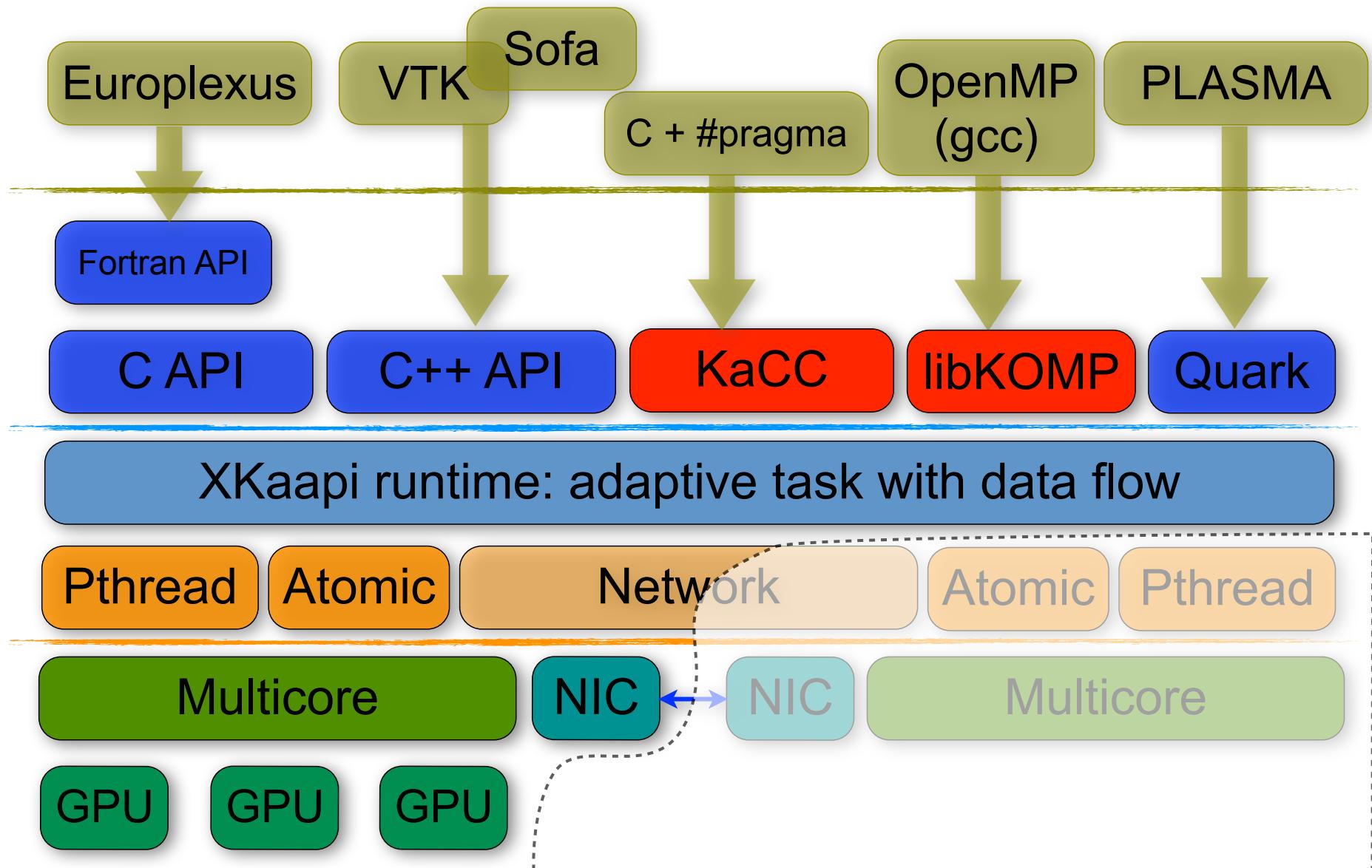
- Evaluation platforms

- AMD48: 4x12 AMD Opteron (6174) cores
- Intel32: 4x8 Intel Xeon (X7560) cores

- Softwares

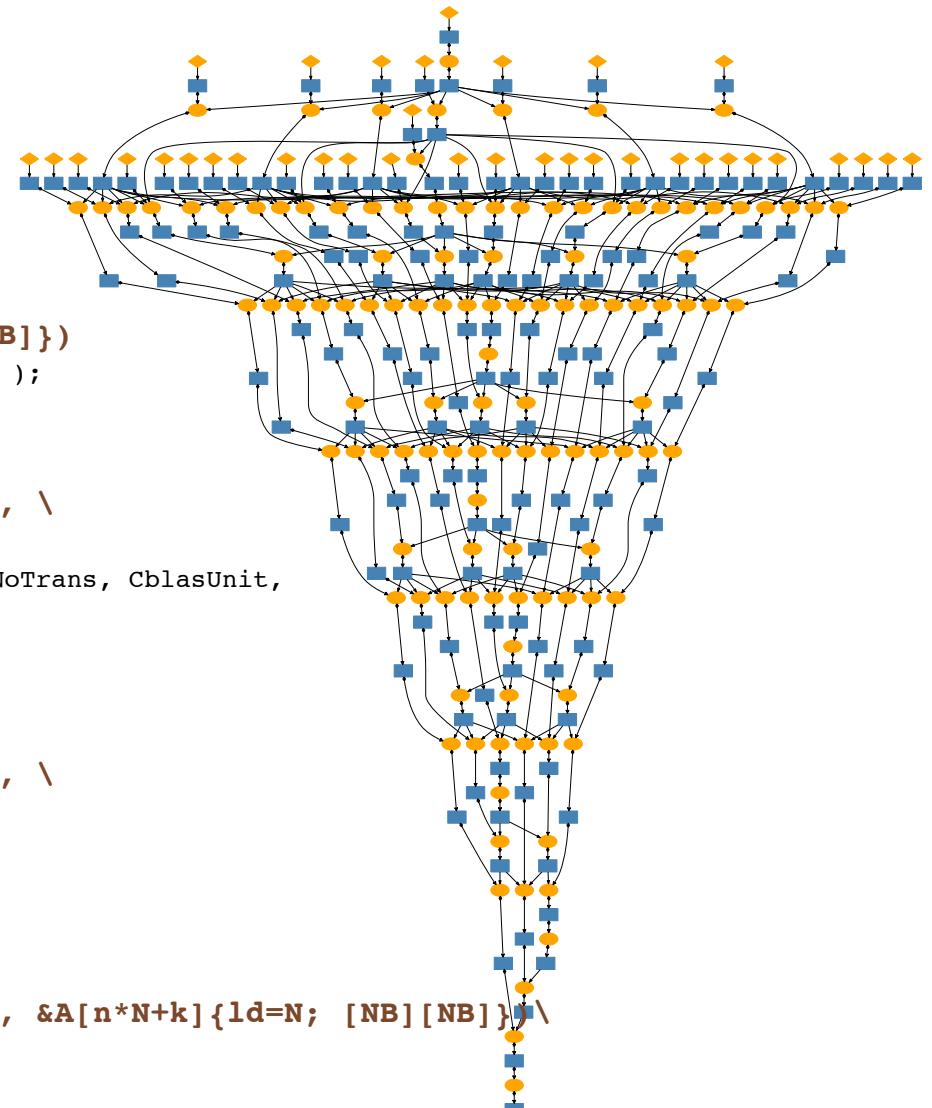
- gcc 4.6.2 + libGOMP
- gcc 4.6.2 + libKOMP
- icc 12.1.2 + Intel OpenMP runtime (KMP)

Software stack



OpenMP version

```
#include <cblas.h>
#include <clapack.h>
void Cholesky( double* A, int N, size_t NB )
{
#pragma omp parallel
    for (size_t k=0; k < N; k += NB)
#pragma omp single
    {
#pragma omp task depend(inout: &A[k*N+k]{ld=N; [NB][NB]})\n
        clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );\n\n
        for (size_t m=k+ NB; m < N; m += NB)\n
        {\n#pragma omp task depend(in: &A[k*N+k]{ld=N; [NB][NB]}, \n
                               inout: &A[m*N+k]{ld=N; [NB][NB]})\n
            cblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit,\n
                          NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );\n
        }\n\n
        for (size_t m=k+ NB; m < N; m += NB)\n
        {\n#pragma omp task depend(in: &A[m*N+k]{ld=N; [NB][NB]}, \n
                               inout: &A[m*N+m]{ld=N; [NB][NB]})\n
            cblas_dsyrk ( CblasRowMajor, CblasLower, CblasNoTrans,\n
                          NB, NB, -1.0, &A[m*N+k], N, 1.0, &A[m*N+m], N );\n\n
            for (size_t n=k+NB; n < m; n += NB)\n
            {\n#pragma omp task depend(in: &A[m*N+k]{ld=N; [NB][NB]}, &A[n*N+k]{ld=N; [NB][NB]}, \n
                                   inout: &A[m*N+n]{ld=N; [NB][NB]})\n
                cblas_dgemm ( CblasRowMajor, CblasNoTrans, CblasTrans,\n
                              NB, NB, NB, -1.0, &A[m*N+k], N, &A[n*N+k], N, 1.0, &A[m*N+n], N );\n
            }\n
        }\n
    }\n
}
```



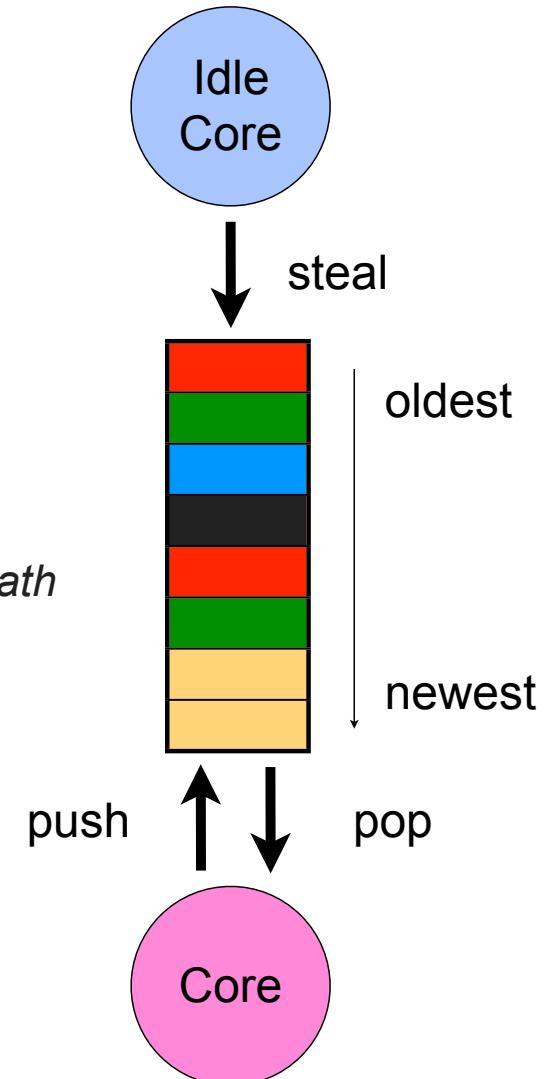
Overhead of task management [Xeon Phi]

- Fibonacci (38) naive recursive computation
- 60 cores of Intel Xeon Phi 5100

#threads	Cilk+ (s)	OpenMP (gcc) (s)	XKaapi (s)
1	33.21	65.64	15.52
10	3.34	33.12	1.58
20	1.66	17.54	0.79
60	0.56	6.30	0.27
120	0.38	3.86	0.18
240	0.37	3.18	0.18

Stealing a task

- A thief thread do
 - ▶ Iteration through the tasks in a victim queue
 - iteration order = creation time order
 - ▶ Computation of data flow dependencies
 - with previously visited tasks
 - ▶ Detection of ready tasks
- Lazy computation of data flow dependencies
 - ▶ In work stealing “theory”, called work first principle
 - overhead is move from the *work* of the program to the *critical path*

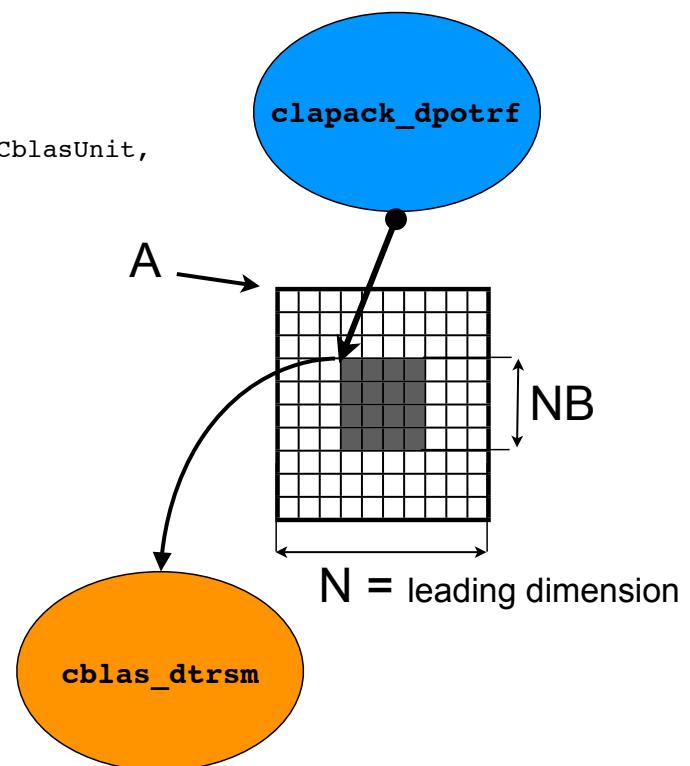




Memory region description

```
#include <cblas.h>
#include <clapack.h>

void Cholesky( double* A, int N, size_t NB )
{
    for (size_t k=0; k < N; k += NB)
    {
        #pragma kaapi task readwrite(&A[k*N+k]{ld=N; [NB][NB]})\n
        clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );\n\n
        for (size_t m=k+ NB; m < N; m += NB)\n
        {\n            #pragma kaapi task read(&A[k*N+k]{ld=N; [NB][NB]}) \\\n            readwrite(&A[m*N+k]{ld=N; [NB][NB]})\n            cblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit,\n                NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );\n        }\n\n
    ...
}
```



False dependencies resolution

- Also called:

- Write after Read
- Writer after Write

- Here =>

- Task1 and Task2 cannot be concurrent
except if 'a' is duplicated
 - also called 'variable renaming'

